expr manual page - Tcl Built-In Commands

tcl.tk/man/tcl/TclCmd/expr.htm

NAME

expr — Evaluate an expression

SYNOPSIS

expr *arg* ?*arg arg ...*?

DESCRIPTION

Concatenates *arg*s (adding separator spaces between them), evaluates the result as a Tcl expression, and returns the value. The operators permitted in Tcl expressions include a subset of the operators permitted in C expressions. For those operators common to both Tcl and C, Tcl applies the same meaning and precedence as the corresponding C operators. Expressions almost always yield numeric results (integer or floating-point values). For example, the expression

```
expr 8.2 + 6
```

evaluates to 14.2. Tcl expressions differ from C expressions in the way that operands are specified. Also, Tcl expressions support non-numeric operands and string comparisons, as well as some additional operators not found in C.

OPERANDS

A Tcl expression consists of a combination of operands, operators, parentheses and commas. White space may be used between the operands and operators and parentheses (or commas); it is ignored by the expression's instructions. Where possible, operands are interpreted as integer values. Integer values may be specified in decimal (the normal case), in binary (if the first two characters of the operand are **0b**), in octal (if the first two characters of the operand are **0o**), or in hexadecimal (if the first two characters of the operand are **0x**). For compatibility with older Tcl releases, an octal integer value is also indicated simply when the first character of the operand is **0**, whether or not the second character is also **o**. If an operand does not have one of the integer formats given above, then it is treated as a floating-point number if that is possible. Floating-point numbers may be specified in any of several common formats making use of the decimal digits, the decimal point **.**, the characters **e** or **E** indicating scientific notation, and the sign characters **+** or **-**. For example, all of the following are valid floating-point numbers: 2.1, 3., 6e4, 7.91e+16. Also recognized as floating point values are the strings **Inf** and **NaN** making use of any case for each character. If no numeric interpretation is possible (note that all literal operands that are not numeric or boolean must be quoted with either braces or with double quotes), then an operand is left as a string (and only a limited set of operators may be applied to it).

Operands may be specified in any of the following ways:

1. As a numeric value, either integer or floating-point.
2. As a boolean value, using any form understood by **string is boolean**.
3. As a Tcl variable, using standard **$** notation. The variable's value will be used as the operand.
4. As a string enclosed in double-quotes. The expression parser will perform backslash, variable, and command substitutions on the information between the quotes, and use the resulting value as the operand
5. As a string enclosed in braces. The characters between the open brace and matching close brace will be used as the operand without any substitutions.
6. As a Tcl command enclosed in brackets. The command will be executed and its result will be used as the operand.
7. As a mathematical function whose arguments have any of the above forms for operands, such as **sin($x)**. See **MATH FUNCTIONS** below for a discussion of how mathematical functions are handled.

Where the above substitutions occur (e.g. inside quoted strings), they are performed by the expression's instructions. However, the command parser may already have performed one round of substitution before the expression processor was called. As discussed below, it is usually best to enclose expressions in braces to prevent the command parser from performing substitutions on the contents.

For some examples of simple expressions, suppose the variable **a** has the value 3 and the variable **b** has the value 6. Then the command on the left side of each of the lines below will produce the value on the right side of the line:

```
expr {3.1 + $a} 6.1
expr {2 + "$a.$b"}      5.6
expr {4*[llength "6 2"]}        8
expr {{word one} < "word $a"}   0
```

OPERATORS

The valid operators (most of which are also available as commands in the **tcl::mathop** namespace; see the **mathop**(n) manual page for details) are listed below, grouped in decreasing order of precedence:

- + ~ !

Unary minus, unary plus, bit-wise NOT, logical NOT. None of these operators may be applied to string operands, and bit-wise NOT may be applied only to integers.

******

Exponentiation. Valid for any numeric operands. The maximum exponent value that Tcl can handle if the first number is an integer > 1 is 268435455.

*** / %**

Multiply, divide, remainder. None of these operators may be applied to string operands, and remainder may be applied only to integers. The remainder will always have the same sign as the divisor and an absolute value smaller than the absolute value of the divisor.

When applied to integers, the division and remainder operators can be considered to partition the number line into a sequence of equal-sized adjacent non-overlapping pieces where each piece is the size of the divisor; the division result identifies which piece the divisor lay within, and the remainder result identifies where within that piece the divisor lay. A consequence of this is that the result of “-57 **/** 10” is always -6, and the result of “-57 **%** 10” is always 3.

+ -
Add and subtract. Valid for any numeric operands.

<< >>
Left and right shift. Valid for integer operands only. A right shift always propagates the sign bit.

< > <= >=
Boolean less, greater, less than or equal, and greater than or equal. Each operator produces 1 if the condition is true, 0 otherwise. These operators may be applied to strings as well as numeric operands, in which case string comparison is used.

== !=
Boolean equal and not equal. Each operator produces a zero/one result. Valid for all operand types.

eq ne
Boolean string equal and string not equal. Each operator produces a zero/one result. The operand types are interpreted only as strings.

in ni
List containment and negated list containment. Each operator produces a zero/one result and treats its first argument as a string and its second argument as a Tcl list. The **in** operator indicates whether the first argument is a member of the second argument list; the **ni** operator inverts the sense of the result.

&
Bit-wise AND. Valid for integer operands only.

^
Bit-wise exclusive OR. Valid for integer operands only.

|
Bit-wise OR. Valid for integer operands only.

&&
Logical AND. Produces a 1 result if both operands are non-zero, 0 otherwise. Valid for boolean and numeric (integers or floating-point) operands only. This operator evaluates lazily; it only evaluates its second operand if it must in order to determine its result.

||
Logical OR. Produces a 0 result if both operands are zero, 1 otherwise. Valid for boolean and numeric (integers or floating-point) operands only. This operator evaluates lazily; it only evaluates its second operand if it must in order to determine its result.

x* ? *y* : *z
If-then-else, as in C. If *x* evaluates to non-zero, then the result is the value of *y*. Otherwise the result is the value of *z*. The *x* operand must have a boolean or numeric value. This operator evaluates lazily; it evaluates only one of *y* or *z*.

See the C manual for more details on the results produced by each operator. The exponentiation operator promotes types like the multiply and divide operators, and produces a result that is the same as the output of the **pow** function (after any type conversions.) All of the binary operators but exponentiation group left-to-right within the same precedence level; exponentiation groups right-to-left. For example, the command

```
expr {4*2 < 7}
```

returns 0, while

```
expr {2**3**2}
```

returns 512.

The **&&**, **||**, and **?:** operators have "lazy evaluation", just as in C, which means that operands are not evaluated if they are not needed to determine the outcome. For example, in the command

```
expr {$v?[a]:[b]}
```

only one of "**[a]**" or "**[b]**" will actually be evaluated, depending on the value of **$v**. Note, however, that this is only true if the entire expression is enclosed in braces; otherwise the Tcl parser will evaluate both "**[a]**" and "**[b]**" before invoking the **expr** command.

MATH FUNCTIONS

When the expression parser encounters a mathematical function such as **sin($x)**, it replaces it with a call to an ordinary Tcl command in the **tcl::mathfunc** namespace. The processing of an expression such as:

```
expr {sin($x+$y)}
```

is the same in every way as the processing of:

```
expr {[tcl::mathfunc::sin [expr {$x+$y}]]}
```

which in turn is the same as the processing of:

```
tcl::mathfunc::sin [expr {$x+$y}]
```

The executor will search for **tcl::mathfunc::sin** using the usual rules for resolving functions in namespaces. Either **::tcl::mathfunc::sin** or **[namespace current]::tcl::mathfunc::sin** will satisfy the request, and others may as well (depending on the current **namespace path** setting).

Some mathematical functions have several arguments, separated by commas like in C. Thus:

```
expr {hypot($x,$y)}
```

ends up as

```
tcl::mathfunc::hypot $x $y
```

See the **mathfunc**(n) manual page for the math functions that are available by default.

TYPES, OVERFLOW, AND PRECISION

All internal computations involving integers are done calling on the LibTomMath multiple precision integer library as required so that all integer calculations are performed exactly. Note that in Tcl releases prior to 8.5, integer calculations were performed with one of the C types *long int* or *Tcl_WideInt*, causing implicit range truncation in those calculations where values overflowed the range of those types. Any code that relied on these implicit truncations will need to explicitly add **int()** or **wide()** function calls to expressions at the points where such truncation is required to take place.
All internal computations involving floating-point are done with the C type *double*. When converting a string to floating-point, exponent overflow is detected and results in the *double* value of **Inf** or **-Inf** as appropriate. Floating-point overflow and underflow are detected to the degree supported by the hardware, which is generally pretty reliable.

Conversion among internal representations for integer, floating-point, and string operands is done automatically as needed. For arithmetic computations, integers are used until some floating-point number is introduced, after which floating-point is used. For example,

```
expr {5 / 4}
```

returns 1, while

```
expr {5 / 4.0}
expr {5 / ( [string length "abcd"] + 0.0 )}
```

both return 1.25. Floating-point values are always returned with a "**.**" or an "**e**" so that they will not look like integer values. For example,

```
expr {20.0/5.0}
```

returns **4.0**, not **4**.

STRING OPERATIONS

String values may be used as operands of the comparison operators, although the expression evaluator tries to do comparisons as integer or floating-point when it can, i.e., when all arguments to the operator allow numeric interpretations, except in the case of the **eq** and **ne** operators. If one of the operands of a comparison is a string and the other has a numeric value, a canonical string representation of the numeric operand value is

generated to compare with the string operand. Canonical string representation for integer values is a decimal string format. Canonical string representation for floating-point values is that produced by the **%g** format specifier of Tcl's **format** command. For example, the commands

```
expr {"0x03" > "2"}
expr {"0y" > "0x12"}
```

both return 1. The first comparison is done using integer comparison, and the second is done using string comparison. Because of Tcl's tendency to treat values as numbers whenever possible, it is not generally a good idea to use operators like **==** when you really want string comparison and the values of the operands could be arbitrary; it is better in these cases to use the **eq** or **ne** operators, or the **string** command instead.

PERFORMANCE CONSIDERATIONS

Enclose expressions in braces for the best speed and the smallest storage requirements. This allows the Tcl bytecode compiler to generate the best code.
As mentioned above, expressions are substituted twice: once by the Tcl parser and once by the **expr** command. For example, the commands

```
set a 3
set b {$a + 2}
expr $b*4
```

return 11, not a multiple of 4. This is because the Tcl parser will first substitute "**$a + 2**" for the variable **b**, then the **expr** command will evaluate the expression "**$a + 2*4**".

Most expressions do not require a second round of substitutions. Either they are enclosed in braces or, if not, their variable and command substitutions yield numbers or strings that do not themselves require substitutions. However, because a few unbraced expressions need two rounds of substitutions, the bytecode compiler must emit additional instructions to handle this situation. The most expensive code is required for unbraced expressions that contain command substitutions. These expressions must be implemented by generating new code each time the expression is executed.

If it is necessary to include a non-constant expression string within the wider context of an otherwise-constant expression, the most efficient technique is to put the varying part inside a recursive **expr**, as this at least allows for the compilation of the outer part, though it does mean that the varying part must itself be evaluated as a separate expression. Thus, in this example the result is 20 and the outer expression benefits from fully cached bytecode compilation.

```
set a 3
set b {$a + 2}
expr {[expr $b] * 4}
```

When the expression is unbraced to allow the substitution of a function or operator, consider using the commands documented in the **mathfunc**(n) or **mathop**(n) manual pages directly instead.

EXAMPLES

Define a procedure that computes an “interesting” mathematical function:

```
proc tcl::mathfunc::calc {x y} {
    expr { ($x**2 - $y**2) / exp($x**2 + $y**2) }
}
```

Convert polar coordinates into cartesian coordinates:

```
# convert from ($radius,$angle)
set x [expr { $radius * cos($angle) }]
set y [expr { $radius * sin($angle) }]
```

Convert cartesian coordinates into polar coordinates:

```
# convert from ($x,$y)
set radius [expr { hypot($y, $x) }]
set angle  [expr { atan2($y, $x) }]
```

Print a message describing the relationship of two string values to each other:

```
puts "a and b are [expr {$a eq $b ? {equal} : {different}}]"
```

Set a variable to whether an environment variable is both defined at all and also set to a true boolean value:

```
set isTrue [expr {
    [info exists ::env(SOME_ENV_VAR)] &&
    [string is true -strict $::env(SOME_ENV_VAR)]
}]
```

Generate a random integer in the range 0..99 inclusive:

```
set randNum [expr { int(100 * rand()) }]
```